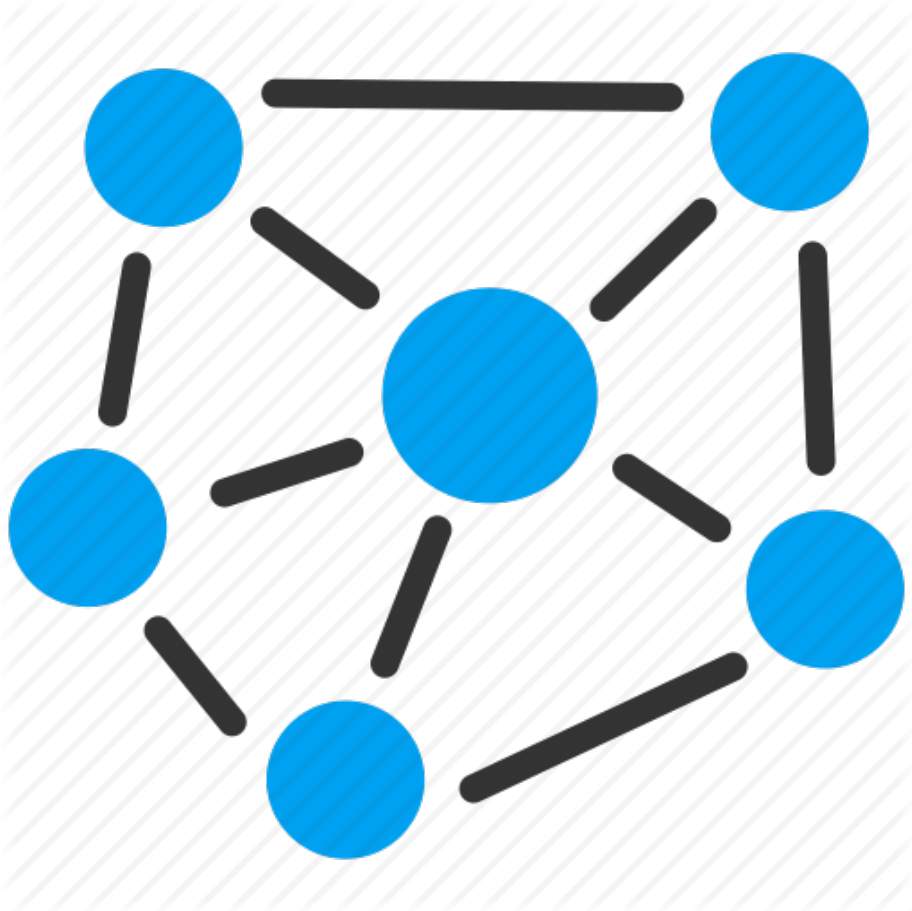
Algoritmi Bread-First and Depth-First su Adjacency Matrix and Adjacency List Graphs

Patrick Jusic

May 21, 2017



Contents

1 Progetto

1.1 Introduzione

L'argomento del progetto è l'implementazione di grafi con le tecniche di *Adjacency Matrix* e *Adjacency Lists*, e l'impego di queste classi in algoritmi di esplorazione della frontiera.
L'obbiettivo è stato il confronto delle prestazioni dei due tipi di grafo, sia a livello strutturale, sia a livello implementativo degli stessi algoritmi.
Un appunto iniziale che vorrei fare è specificare che questo progetto è stato sviluppato durante il corso, a cavallo tra lo studio degli alberi e delle tabelle hash, così da intendere che alcune mancanze o affermazioni imprecise potrebbero essere motivate dall'impreparazione su argomenti, come ad esempio i tipi di dato astratto, che non sono ancora stati trattati.
Con questa premessa, prima di entrare nella descrizione delle parti implementate, vorrei sottolineare ciò che mi ha più colpito nella trattazione dei grafi, cioè l' elevato grado di astrazione che caratterizza l'implementazione di questa struttura.
Infatti se inizialmente può venir naturale pensare ad un implementazione simile a quella seguita negli alberi, risulta poi molto più semplice a livello di sviluppo e concettuale, aumentare il grado di astrazione e gestire questa struttura con una struttura più semplice, come ad esempio una matrice, che non fa uso di puntatori. Questa tecnica risulta, infatti, implementativamente più semplice rispetto a quella mediante l'uso di liste.
Nel progetto non è stata trattata l'implementazione di algoritmi per la ricerca di un elemento, classici dell'argomento, che si basano sull'ottimizzazione della ricerca attraverso il percorso più breve possibile. Qui si apre una finestra molto ampia che meriterebbe un intero progetto al riguardo e qualche altro centinaio di righe di codice per poterne affrontare tutti gli aspetti.
Entrambe le classi implementate sono corredate di metodi che le rendono utilizzabili non solo nella suddetta applicazione, ma già pronte ad essere incorporate con nuovi metodi per diverse applicazioni. Sono implementati metodi per l'aggiunta di collegamenti tra i nodi, la loro rimozione, il controllo e la stampa. Non è stata prevista la possibilità di mettere un limite al numero dei nodi e dei loro collegamento.

1.2 Adjacency Matrix

L' implementazione con la matrice di adiacenza si basa come precedentemente sottolineato sulla creazione di una matrice rappresentante il grafo, associandone i nodi agli indici di righe e colonne.
Il parametro di creazione della matrice è `vertexCount` che rappresenta il numero di nodi del grafo. Viene quindi generata una matrice vuota di $vertexCount^2$ elementi, ognuno settato a 0.
Successivamente, tramite i metodi implementati nella classe, si procede alla creazione degli `Edges`, i collegamenti tra i grafi, rappresentati con il numero 1.

Quindi, in corrispondenza dell'indice riga *i-esimo* e dell'indice colonna *j-esimo*, si trova l'1 che rappresenta il collegamento tra il nodo *i-esimo* e il nodo *j-esimo* del grafo.
Questo è un collegamento `Diretto`, caratteristica che può caratterizzare l'intero grafo, e significa che l'edge originato dal nodo *i-esimo* con destinazione il nodo *j-esimo*, non può essere percorso in senso opposto. In questo caso si parlerebbe di collegamento `Indiretto`, che risulterebbe in un 1 nella matrice in corrispondenza tra l'indice della riga *j-esima* e l'indice della colonna *i-esima*. Intuitivamente risulta che se il grafo è completamente `Indiretto` è rappresentato da una matrice simmetrica.
L'uso di 1 e 0 è molto utile per rappresentare la presenza di un grafo, ma può essere rimpiazzato con l'utilizzo di valori $\epsilon[0, 1]$, così da rappresentare un grafo con collegamenti pesati, utile in applicazioni come ad esempio reti neurali.

1.3 Adjacency List

L' implementazione con lista di adiacenza ricorda maggiormente il tipo di struttura ad albero perchè presenta in modo esplicito delle struct `Node`, ognuno dei quali contiene il valore corrispondente all'indice del nodo rappresentato, ed un puntatore all'elemento successivo.
L'idea è quella di rappresentare un array di puntatori, della lunghezza di `vertexCount`, il quale riferisce all'indice dell'elemento *i-esimo* il nodo *i-esimo* del grafo, elemento dal quale parte il puntatore ad una lista contenente tutti i nodi della frontiera del nodo *i-esimo*. La frontiera è l'insieme dei nodi collegati al nodo in questione. Tale struttura può ricordare idealmente quella del vettore di bucket discusso nella Tabelle Hash.
Quindi la generazione del grafo si compone della creazione dell'array, e dell'inizializzazione a NULL di tutti i suoi puntatori. Successivamente con un metodo della classe si aggiungono elementi alla lista corrispondente, anche qui con il principio di collegamento `Diretto` o `Indiretto`. Risulta evidente la maggior complessità di sviluppo di questa struttura, ma in realtà l'implementazione dei metodi ad essa relativi non risultano poi particolarmente ostici.

1.4 Algoritmi

L'idea dietro gli algoritmi implementati è quella di esplorare il grafo, attraverso due differenti approcci.
Per poter tener traccia dell'esplorazione si concettualizza con la colorazione dei nodi il loro stato. La dichiarazione di tale concetto è svolta attraverso l'*enum* `VertexState`, che elenca i tre diversi stati possibili cioè:

- `White`: questo stato descrive il nodo come mai visitato
- `Grey`: questo stato descrive il nodo in fase di esplorazione, cioè quando la sua frontiera è in fase di esplorazione

- `Black`: questo stato descrive il nodo in post-esplorazione, quando cioè tutta la sua frontiera è stata esplorata

L'algoritmo di esplorazione si dirà concluso quando tutti i nodi del grafo saranno stati colorati di nero, ad eccezione di quei nodi che non sono collegati a nessun altro nodo e non hanno modo di essere raggiunti, i quali risuleranno inesistenti per il grafo.
La complessità computazionale di entrambi gli algoritmi è $\theta(n)$, dato che in entrambi i casi il grafo viene completamente esplorato, quindi bisognerà visitare tutti gli n nodi presenti nella sua struttura.

1.4.1 Depth-First Algorithm

L'algoritmo Depth-First è costruito in modo tale da inseguire la profondità del grafo. Come dice il nome infatti questo algoritmo esplora il grafo puntando al fondo di esso, esplorando via via ogni nodo mancante.
Perciò verrà passato come parametro `start`, cioè il nodo da cui cominciare. Una volta messo il suo stato a grigio con un *for* si comincia ad esplorare la sua frontiera. Ricorsivamente lo stesso procedimento verrà applicata al primo nodo incontrato nella frontiera del nodo inziale, che verrà anch'esso messo a grigio.
Il nodo iniziale viene colorato di nero soltanto quando tutta la sua frontiera sarà esplorata, perciò quando tutti i suoi figli, e quindi i figli dei suoi figli, e così via, saranno già stati esplorati.
Quindi l'approccio di questo algoritmo è quello di esplorare il grafo seguendo la sua lunghezza puntando sempre alle foglie, quei nodi cioè che non hanno figli.

1.4.2 Breadth-First Algorithm

L'algoritmo Breadth-First, a differenza del suo collega procede per gradi, esplorando man mano l'intera frontiera del nodo che in questa fase è colorato di grigio.
Quindi come nel caso precedente, all'interno di un ciclo *for*, tutti i nodi collegati al nodo in questione vengono colorati di grigio, ma differenza del Depth-First, lo stato del nodo la cui frontiera è stata completamente visitata è messo a nero, ben prima quindi che lo diventino i suoi figli, e i figli dei figli, i quali saranno successivamenti ricorsivamente sottoposti allo stesso processo.
Perciò in questo caso le foglie sono raggiunte solamente quando tutti i figli del nodo di partenza sono già stati esplorati e colorati di nero.

1.5 Confronto tra Algoritmi

Intuitivamente il Breadth-First risulta più performante, specie per quanto riguarda la ricerca di un elemento. In realtà le performance dei due algoritmi sono profondamente condizionate dalla struttura del grafo, come verrà poi illustrato nella sezione `Risultati`.
In particolare il Breadth-First può risultare inefficiente se i nodi cercati sono in

fondo al grafo, le cosiddette foglie, mentre il discorso opposto vale per il Depth-First nel momento in cui il nodo cercato è l'ultimo dei figli del nodo iniziale, in quanto dev'essere esplorata prima gran parte del grafo.
Per quanto riguarda la ricerca bisognerebbe perciò differenziare casi peggiori e casi migliori contemplando una complessità computazionale che può andare da $\theta(1)$ fino a $\theta(n)$, inversamente in base all'algoritmo applicato.
Non avendo sviluppato il discorso di ricerca di un elemento questo discorso decade in parte, cioè che la struttura del grafo dovrebbe condizionare un po' meno il *tempo effettivo di esecuzione*, avendo entrambi gli algoritmi complessità computazionale $\theta(n)$, ed essendo impiegati con lo stesso fine. In realtà ci sarà modo di esporre i risultati e discuterne il significato.

2 Codice

Sono di seguito riportati tutti i codici che compongono il progetto suddivisi in base agli header files, cioè i files linkati, e il programma principale dove sono state svolte le prove.
Nel programma principale è stata implementata la funzione `decorateGraph`, con lo scopo di generare in maniera randomica dei grafi, con entrambi i modi di sviluppo presentati, con lo stesso numeri di nodi e collegamenti, per avere dei confronti il più affidabili possibile.
Oltre ai files header contenenti le classi che implementano i grafi, sono stati linkati altri due file, uno contenente una classe `Timer`, che sfrutta la libreria `ctime` per calcolare il tempo effettivo di esecuzione degli algoritmi in microsecondi, e l'altro contente l'implementazione di una `Queue`, con la struttura e metodi discussi durante il corso, utilizzata nel Breadth-First della Matrice di Adiacenza.

2.1 Headers

2.1.1 Matrice di Adiacenza

```
#ifndef MATRIX
#define MATRIX

#include <iostream>
using namespace std;

#include "Queue.h"

class Graph {
  private:
    bool** adjacencyMatrix;
    int vertexCount;
    enum VertexState { White, Grey, Black };
  public:
    Graph(int vertexCount) {
      this->vertexCount = vertexCount;
      cout << "vertexCount: " << vertexCount << endl;
      adjacencyMatrix = new bool*[vertexCount];
      cout << "adjacencyMatrix : " << adjacencyMatrix << endl;
      for (int i = 0; i < vertexCount; i++) {
        adjacencyMatrix[i] = new bool[vertexCount];
        for (int j = 0; j < vertexCount; j++) {
                adjacencyMatrix[i][j] = false;
        }
      }
    }

    void printAdjacencyMatrix() {
      for (int i = 0; i < vertexCount; i++) {
        cout << "adjacencyMatrix[" << i << "]: " << adjacencyMatrix
    [i] << "    ";
        for (int j = 0; j < vertexCount; j++) {
```

```
32              adjacencyMatrix[i][j] ? cout << 1 << " " : cout << 0 <<
    " ";
33          }
34          cout << endl;
35        }
36        cout << endl;
37      }
38
39      void addDirectEdge(int i, int j) {
40        if (i >= 0 && i < vertexCount && j >= 0 && j < vertexCount) {
41          adjacencyMatrix[i][j] = true;
42        }
43      }
44
45      void addUnidirectEdge(int i, int j) {
46        if (i >= 0 && i < vertexCount && j >= 0 && j < vertexCount) {
47          adjacencyMatrix[i][j] = true;
48          adjacencyMatrix[j][i] = true;
49        }
50      }
51
52      void removeEdge(int i, int j) {
53        if (i >= 0 && i < vertexCount && j > 0 && j < vertexCount) {
54          adjacencyMatrix[i][j] = false;
55          adjacencyMatrix[j][i] = false;
56        }
57      }
58
59      bool isEdge(int i, int j) {
60        if (i >= vertexCount || j >= vertexCount) throw
    invalid_argument("Vertex doesn't exist");
61        if (i >= 0 && j > 0)
62          return adjacencyMatrix[i][j];
63        else
64          return false;
65      }
66
67      bool checkEdge(int i, int j) {
68        try {
69          return adjacencyMatrix[i][j];
70        }
71        catch(invalid_argument &e) {
72          cerr << e.what() << endl;
73          return false;
74        }
75      }
76
77      void DFS(int start) {
78      VertexState *state = new VertexState[vertexCount];
79      for (int i = 0; i < vertexCount; i++) state[i] = White;
80      runDFS(start, state);
81      delete [] state;
82      }
83      void runDFS(int u, VertexState state[]) {
84        state[u] = Grey;
85        for (int v = 0; v < vertexCount; v++) {
86          if (checkEdge(u, v) && state[v] == White) {
```

```
87            runDFS(v, state);
88          }
89        }
90        state[u] = Black;
91      }
92
93
94      void BFS(int start) {
95        VertexState *state = new VertexState[vertexCount];
96        for (int i = 0; i < vertexCount; i++)
97          state[i] = White;
98        state[start] = Grey;
99        TQueue Q = queue_create(vertexCount);
100       queue_add(&Q, start);
101       runBFS(start, state, &Q);
102       delete [] state;
103     }
104     int runBFS(int u, VertexState state[], TQueue* Q) {
105       if(queue_is_empty(Q)) return 0;
106       for(int v = 0; v < vertexCount; v++) {
107         if(checkEdge(u,v) && state[v] == White) {
108           state[v] = Grey;
109           queue_add(Q, v);
110         }
111       }
112       state[u] = Black;
113       queue_remove(Q);
114       for(int j = Q->a[Q->front]; Q->front != Q->back; j = Q->a[j
    +1]) {
115         runBFS(j, state, Q);
116       }
117       return 1;
118     }
119
120     ~Graph() {
121       for (int i = 0; i < vertexCount; i++)
122         delete[] adjacencyMatrix[i];
123         delete[] adjacencyMatrix;
124     }
125 };
126
127 #endif
```

2.1.2 Lista di Adiacenza

```
1  #ifndef LIST
2  #define LIST
3
4  #include <iostream>
5  #include <cstdlib>
6  using namespace std;
7
8  struct AdjListNode {
9    int dest;
10   struct AdjListNode* next;
11 };
12
13 struct AdjList {
```

```
14    struct AdjListNode *head;
15  };
16
17  class GraphList {
18    private:
19      int vertexCount;
20      struct AdjList* array;
21      enum VertexState { White, Grey, Black };
22    public:
23      GraphList(int vertexCount) {
24        this->vertexCount = vertexCount;
25        array = new AdjList[vertexCount];
26        for (int i = 0; i < vertexCount; ++i)
27          array[i].head = NULL;
28      }
29
30      AdjListNode* newAdjListNode(int dest) {
31        AdjListNode* newNode = new AdjListNode;
32        newNode->dest = dest;
33        newNode->next = NULL;
34        return newNode;
35      }
36
37      int checkEdge(int src, int dest) {
38        AdjListNode* pCrawl = array[src].head;
39        if(!pCrawl) return 0;
40        while(pCrawl) {
41          if(pCrawl->dest == dest) break;
42          pCrawl = pCrawl->next;
43          if(!pCrawl) return 0;
44        }
45        return -1;
46      }
47
48      void addDirectEdge(int src, int dest) {
49        if(!checkEdge(src, dest)) {
50          AdjListNode* newNode = newAdjListNode(dest);
51          newNode->next = array[src].head;
52          array[src].head = newNode;
53        }
54      }
55
56      void addUnidirectEdge(int src, int dest) {
57        if(!checkEdge(src, dest)) {
58          AdjListNode* newNode = newAdjListNode(dest);
59          newNode->next = array[src].head;
60          array[src].head = newNode;
61          newNode = newAdjListNode(src);
62          newNode->next = array[dest].head;
63          array[dest].head = newNode;
64        }
65      }
66
67      void removeEdge(int src, int dest) {
68        AdjListNode* pCrawl = array[src].head;
69        while(pCrawl->next->dest != dest) pCrawl = pCrawl->next;
70        pCrawl->next = pCrawl->next->next;
```

```
71      }
72
73      void printAdjacencyList() {
74        for (int v = 0; v < vertexCount; ++v) {
75          AdjListNode* pCrawl = array[v].head;
76          cout << "Adjacency list of vertex " << v << "\n head ";
77          while(pCrawl) {
78            cout << "-> " << pCrawl->dest;
79            pCrawl = pCrawl->next;
80          }
81          cout << endl;
82        }
83      }
84
85
86      void DFS(int start) {
87        VertexState *state = new VertexState[vertexCount];
88        for (int i = 0; i < vertexCount; i++) state[i] = White;
89        runDFS(start, state);
90        delete [] state;
91      }
92      void runDFS(int u, VertexState state[]) {
93        state[u] = Grey;
94        AdjListNode* pCrawl = array[u].head;
95        while(pCrawl) {
96          if(state[pCrawl->dest] == White)
97            runDFS(pCrawl->dest, state);
98          pCrawl = pCrawl->next;
99        }
100     }
101
102
103     void BFS(int start) {
104       VertexState *state = new VertexState[vertexCount];
105       for (int i = 0; i < vertexCount; i++) state[i] = White;
106       state[start] = Grey;
107       runBFS(start, state);
108       delete [] state;
109     }
110     void runBFS(int u, VertexState state[]) {
111       AdjListNode* pCrawl = array[u].head;
112       while(pCrawl) {
113         if(state[pCrawl->dest] == White) {
114           state[pCrawl->dest] = Grey;
115         }
116         pCrawl = pCrawl->next;
117       }
118       state[u] = Black;
119       pCrawl = array[u].head;
120       while(pCrawl) {
121         if(state[pCrawl->dest] != Black)
122           runBFS(pCrawl->dest, state);
123         pCrawl = pCrawl->next;
124       }
125     }
126 };
127
```

```
#endif
```

2.1.3 Queue

```
#ifndef QUEUE
#define QUEUE

#include <iostream>
#include <stdlib.h>
using namespace std;

/* Definizione della struttura dati */
struct SQueue {
  int n;
  int front;
  int back;
  int capacity;
  int* a ;
};
typedef struct SQueue TQueue ;

/* Crea una coda */
TQueue queue_create (int capacity) {
  TQueue s;
  s.n = 0;
  s.front = 0;
  s.back = 0;
  s.capacity = capacity;
  s.a = (int*)malloc(sizeof(int)*capacity);
  // s.a = (int*)malloc(capacity);
  // assert(s.a != NULL);
  return s;
}

/* Distrugge una coda*/
void queue_destroy (TQueue* queue) {
  queue->n = 0;
  queue->capacity = 0;
  free(queue->a);
  queue->a = NULL;
}

/* Accoda un elemento*/
void queue_add(TQueue* queue, int x) {
  queue->a[queue->back] = x;
  queue->back = (queue->back+1) % queue->capacity;
  queue->n++;
}

/* Preleva un elemento*/
int queue_remove(TQueue* queue) {
  int x = queue->a[queue->front];
  queue->front = (queue->front+1) % queue->capacity;
  queue->n--;
  return x;
}

/* Primo elemento*/
```

```
int queue_front(TQueue* queue) {
  return queue->a[queue->front];
}

/* Verifica se la coda    vuota*/
bool queue_is_empty(TQueue* queue) {
  return queue->n == 0;
}

/* Verifica se la coda    piena*/
bool queue_is_full(TQueue* queue) {
  return queue->n == queue->capacity;
}


#endif
```

2.1.4 Timer

```
#ifndef TIMER
#define TIMER

#include <ctime>


class Timer
{
public:
    Timer() { clock_gettime(CLOCK_REALTIME, &beg_); }

    double elapsed() {
        clock_gettime(CLOCK_REALTIME, &end_);
        return end_.tv_sec - beg_.tv_sec +
            (end_.tv_nsec - beg_.tv_nsec) / 1000000000.;
    }

    void reset() { clock_gettime(CLOCK_REALTIME, &beg_); }

private:
    timespec beg_, end_;
};


#endif
```

2.2 Main

```
#include <iostream>
#include <stdlib.h>

using namespace std;

#include "AdjacencyMatrix.h"
#include "AdjacencyList.h"
#include "Timer.h"

```

```
11 void decorateGraph(int vertexes, Graph* g=NULL, GraphList* gl=NULL)
       {
12   int src, dest = 0;
13   int edges = rand() % vertexes*vertexes + 1;
14   for(int i=0; i < edges; i++) {
15     src = rand() % vertexes;
16     dest = rand() % vertexes;
17     if(g) g->addDirectEdge(src, dest);
18     if(gl) gl->addDirectEdge(src, dest);
19   }
20 }
21
22
23
24
25 int main() {
26   srand(time(NULL));
27   int vertexes = rand() % 10 + 100;
28   Graph graph(vertexes);
29   GraphList graphl(vertexes);
30   decorateGraph(vertexes, &graph, &graphl);
31
32
33   cout << "Adjacency Matrix Graph Implementation" << endl;
34   graph.printAdjacencyMatrix();
35   cout << "Breadth-First Search Algorithm" << endl;
36   Timer tmr;
37   graph.BFS(0);
38   double t = tmr.elapsed();
39   cout << "Breadth-First Algorithm Time: " << t << endl;
40
41   cout << "Depth-First Search Algorithm" << endl;
42   tmr.reset();
43   graph.DFS(0);
44   t = tmr.elapsed();
45   cout << "Depth-First Algorithm Time: " << t << endl << endl <<
       endl << endl;
46
47
48
49   cout << "Adjacency List Graph Implementation" << endl;
50   graphl.printAdjacencyList();
51   cout << endl <<"Breadth-First Search Algorithm" << endl;
52   tmr.reset();
53   graphl.BFS(0);
54   t = tmr.elapsed();
55   cout << "Breadth-First Algorithm Time: " << t << endl;
56
57   cout << "Depth-First Search Algorithm" << endl;
58   tmr.reset();
59   graph.DFS(0);
60   t = tmr.elapsed();
61   cout << "Depth-First Algorithm Time: " << t << endl;
62 }
```

3 Risultati

In questa sezione vengono riportati gli output del software sui quali svolgere le adeguate considerazioni e le analisi statistiche risultanti.
Non sono sicuramente stati analizzati tutti i casi possibili, ma la speranza è che i dati portati permettano un'analisi pertinente del problema.
Ci tengo a sottolineare che i tempi effettivi di esecuzione riportati possono essere condizionati da una mancata ottimizzazione degli algoritmi in questione, ragion per cui con una differente codifica potrebbero essere smentiti. In particolar l'utilizzo di una coda nel Breadth-First, sviluppato nella matrice di adiacenza, applicazione basata sulla letteratura esistente, mi suggerisce una possibile miglioria, rispetto agli altri algoritmi che non ne fanno uso e si risparmiano la generazione e la gestione di un'altra struttura dati che grava sul tempo effettivo di esecuzione.

3.1 Output

Sono stati riportati 5 esempi di output con un numero di nodi componenti la struttura crescente, scelti perchè sembrano portare spunti interessanti, analizzandone i tempi effettivi di esecuzioni.

3.1.1 5 Nodi

In questa configurazione, con un basso numero di nodi ed un discreto numero percentuale di collegamenti tra di essi, in entrambe le implementazioni il Depth-First vanta quasi un ordine di grandezza inferiore relativo al suo tempo effettivo di esecuzione rispetto a quello del Breadth-First.
I tempi del Depth-First sono molto vicini, mentre guadagna qualcosa il Breadth-First della lista di adiacenza implementato senza utilizzo di una coda.

```
Adjacency Matrix Graph Implementation
adjacencyMatrix[0]: 0x13d4060    1 1 0 0 1
adjacencyMatrix[1]: 0x13d4080    1 1 1 1 0
adjacencyMatrix[2]: 0x13d40a0    0 1 0 0 1
adjacencyMatrix[3]: 0x13d40c0    0 1 1 1 0
adjacencyMatrix[4]: 0x13d40e0    1 1 0 0 0
Breadth-First Algorithm Time: 4.432e-06
Depth-First Algorithm Time: 9.97e-07

Adjacency List Graph Implementation
Adjacency list of vertex 0
 head -> 1-> 4-> 0
Adjacency list of vertex 1
 head -> 2-> 3-> 1-> 0
Adjacency list of vertex 2
 head -> 1-> 4
Adjacency list of vertex 3
 head -> 2-> 1-> 3
Adjacency list of vertex 4
 head -> 0-> 1
```

```
Breadth-First Algorithm Time: 1.009e-06
Depth-First Algorithm Time: 9.34e-07
```

3.1.2 9 Nodi

Con 9 nodi si avvicinano i tempi degli algoritmi di esplorazione per la lista di adiacenza, mentre restano ancora lontani quelli per la matrice di adiacenza dove il Depth-First è ancora nettamente più rapido.

```
Adjacency Matrix Graph Implementation
adjacencyMatrix[0]: 0x13f8080    1 0 1 1 0 0 0 1 0
adjacencyMatrix[1]: 0x13f80a0    1 0 0 0 0 1 0 1 0
adjacencyMatrix[2]: 0x13f80c0    1 0 1 0 0 1 0 1 0
adjacencyMatrix[3]: 0x13f80e0    1 0 0 0 1 0 1 1 1
adjacencyMatrix[4]: 0x13f8100    1 1 1 1 0 0 1 0 0
adjacencyMatrix[5]: 0x13f8120    0 0 0 1 0 1 1 0 0
adjacencyMatrix[6]: 0x13f8140    1 0 1 0 1 1 0 0 1
adjacencyMatrix[7]: 0x13f8160    0 1 0 0 0 0 1 0 1
adjacencyMatrix[8]: 0x13f8180    1 0 1 1 0 0 1 1 1
Breadth-First Algorithm Time: 1.2055e-05
Depth-First Algorithm Time: 5.185e-06

Adjacency List Graph Implementation
Adjacency list of vertex 0
 head -> 7-> 3-> 0-> 2
Adjacency list of vertex 1
 head -> 5-> 7-> 0
Adjacency list of vertex 2
 head -> 0-> 5-> 2-> 7
Adjacency list of vertex 3
 head -> 7-> 6-> 4-> 0-> 8
Adjacency list of vertex 4
 head -> 0-> 3-> 2-> 6-> 1
Adjacency list of vertex 5
 head -> 6-> 5-> 3
Adjacency list of vertex 6
 head -> 4-> 2-> 8-> 5-> 0
Adjacency list of vertex 7
 head -> 6-> 1-> 8
Adjacency list of vertex 8
 head -> 7-> 3-> 0-> 6-> 8-> 2
Breadth-First Algorithm Time: 4.054e-06
Depth-First Algorithm Time: 4.967e-06
```

3.1.3 10 Nodi

Con solamente un nodo in più i tempi di esecuzione si riducono ancora ma mantengono la stessa proporzione del grafo con 9 nodi.

```
Adjacency Matrix Graph Implementation
adjacencyMatrix[0]: 0x1754090    0 0 0 1 1 0 0 1 0 1
adjacencyMatrix[1]: 0x17540b0    1 0 0 1 0 0 0 1 1 1
adjacencyMatrix[2]: 0x17540d0    1 0 1 0 1 0 0 1 1 1
adjacencyMatrix[3]: 0x17540f0    0 0 1 0 1 1 1 1 1 1
adjacencyMatrix[4]: 0x1754110    1 0 0 1 1 1 1 1 1 1
```

```
adjacencyMatrix[5]: 0x1754130    1 1 0 1 1 0 0 1 0 1
adjacencyMatrix[6]: 0x1754150    0 0 0 0 0 0 0 1 1 1
adjacencyMatrix[7]: 0x1754170    0 0 0 1 0 0 0 0 0 0
adjacencyMatrix[8]: 0x1754190    1 1 0 0 0 1 0 1 0 0
adjacencyMatrix[9]: 0x17541b0    0 1 0 0 0 1 0 0 0 1
Breadth-First Algorithm Time: 8.469e-06
Depth-First Algorithm Time: 3.724e-06

Adjacency List Graph Implementation
Adjacency list of vertex 0
 head -> 9-> 4-> 7-> 3
Adjacency list of vertex 1
 head -> 3-> 0-> 9-> 7-> 8
Adjacency list of vertex 2
 head -> 9-> 4-> 7-> 8-> 2-> 0
Adjacency list of vertex 3
 head -> 6-> 8-> 7-> 9-> 5-> 4-> 2
Adjacency list of vertex 4
 head -> 3-> 9-> 6-> 4-> 8-> 5-> 0-> 7
Adjacency list of vertex 5
 head -> 9-> 0-> 7-> 4-> 1-> 3
Adjacency list of vertex 6
 head -> 8-> 9-> 7
Adjacency list of vertex 7
 head -> 3
Adjacency list of vertex 8
 head -> 5-> 7-> 0-> 1
Adjacency list of vertex 9
 head -> 5-> 9-> 1
Breadth-First Algorithm Time: 3.029e-06
Depth-First Algorithm Time: 2.626e-06
```

3.1.4 14 Nodi

Con 14 nodi si nota una stabilizzazione sui tempi di esecuzione, c'è un miglioramento del Breadth-First rispetto al Depth-First, per la lista di adiacenza, che fin qui era sempre stato più lento.

```
Adjacency Matrix Graph Implementation
adjacencyMatrix[0]: 0xc530b0    0 0 1 0 0 0 0 0 0 1 1 0 0 0
adjacencyMatrix[1]: 0xc530d0    0 0 0 0 0 0 0 1 0 1 0 1 0 0
adjacencyMatrix[2]: 0xc530f0    1 1 0 0 0 1 0 0 0 1 0 1 0 0
adjacencyMatrix[3]: 0xc53110    0 1 0 0 1 0 0 0 1 0 0 0 1 0
adjacencyMatrix[4]: 0xc53130    0 0 0 1 0 0 0 0 0 0 0 0 1 0
adjacencyMatrix[5]: 0xc53150    0 1 0 0 0 1 0 0 0 0 1 0 1 1
adjacencyMatrix[6]: 0xc53170    0 0 0 1 0 0 0 0 0 1 0 0 0 1
adjacencyMatrix[7]: 0xc53190    0 0 0 0 1 0 0 1 1 0 1 0 1 0
adjacencyMatrix[8]: 0xc531b0    0 0 0 1 0 0 0 0 0 0 0 0 0 0
adjacencyMatrix[9]: 0xc531d0    0 1 0 0 1 0 0 0 0 0 1 0 0 1
adjacencyMatrix[10]: 0xc531f0   1 0 0 0 1 0 0 0 0 0 0 0 0 0
adjacencyMatrix[11]: 0xc53210   0 0 0 0 1 0 1 1 0 0 1 0 0 0
adjacencyMatrix[12]: 0xc53230   0 1 1 1 0 0 0 0 0 0 0 0 1 1
adjacencyMatrix[13]: 0xc53250   0 0 0 1 0 0 0 1 0 0 0 0 1 1
Breadth-First Algorithm Time: 7.668e-06
Depth-First Algorithm Time: 4.274e-06

```

```
Adjacency List Graph Implementation
Adjacency list of vertex 0
 head -> 10-> 2-> 9
Adjacency list of vertex 1
 head -> 11-> 7-> 9
Adjacency list of vertex 2
 head -> 5-> 9-> 11-> 1-> 0
Adjacency list of vertex 3
 head -> 12-> 1-> 8-> 4
Adjacency list of vertex 4
 head -> 12-> 3
Adjacency list of vertex 5
 head -> 1-> 13-> 5-> 10-> 12
Adjacency list of vertex 6
 head -> 3-> 13-> 9
Adjacency list of vertex 7
 head -> 4-> 7-> 8-> 10-> 12
Adjacency list of vertex 8
 head -> 3
Adjacency list of vertex 9
 head -> 10-> 13-> 4-> 1
Adjacency list of vertex 10
 head -> 4-> 0
Adjacency list of vertex 11
 head -> 10-> 6-> 4-> 7
Adjacency list of vertex 12
 head -> 13-> 1-> 3-> 2-> 12
Adjacency list of vertex 13
 head -> 13-> 12-> 7-> 3
Breadth-First Algorithm Time: 2.369e-06
Depth-First Algorithm Time: 2.547e-06
```

3.1.5 21 Nodi

Con 21 nodi si conferma l'intuizione precedente perchè il Breadth-First si dimostra più veloce rispetto al Depth-First, sempre per quanto riguarda la lista di adiacenza.

```
Adjacency Matrix Graph Implementation
adjacencyMatrix[0]: 0x22d30e0   0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
     0 1 1
adjacencyMatrix[1]: 0x22d3100   0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0
     0 0 0
adjacencyMatrix[2]: 0x22d3120   1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0
     0 0 1
adjacencyMatrix[3]: 0x22d3140   0 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 0 0
     0 0 1
adjacencyMatrix[4]: 0x22d3160   0 1 0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 1
     0 0 0
adjacencyMatrix[5]: 0x22d3180   0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0
     0 0 0
adjacencyMatrix[6]: 0x22d31a0   0 0 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0
     0 0 0
adjacencyMatrix[7]: 0x22d31c0   0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0
     1 0 0
```

```
adjacencyMatrix[8]: 0x22d31e0   0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
    0 0 0
adjacencyMatrix[9]: 0x22d3200   0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0
    0 0 1
adjacencyMatrix[10]: 0x22d3220  0 0 0 0 1 0 0 1 0 1 1 0 1 0 0 1 0 0
    0 0 1
adjacencyMatrix[11]: 0x22d3240  0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1
    0 0 0
adjacencyMatrix[12]: 0x22d3260  0 1 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 1
    0 0 0
adjacencyMatrix[13]: 0x22d3280  0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
    0 0 0
adjacencyMatrix[14]: 0x22d32a0  1 1 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0
    0 1 0
adjacencyMatrix[15]: 0x22d32c0  0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
    1 1 0
adjacencyMatrix[16]: 0x22d32e0  0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1
    0 0 0
adjacencyMatrix[17]: 0x22d3300  0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0
    1 0 0
adjacencyMatrix[18]: 0x22d3320  1 1 0 1 0 0 1 0 0 1 1 0 0 0 0 0 0 0
    1 0 0
adjacencyMatrix[19]: 0x22d3340  0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0
    0 0 0
adjacencyMatrix[20]: 0x22d3360  0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0
    0 0 1
Breadth-First Algorithm Time: 1.1e-05
Depth-First Algorithm Time: 7.023e-06

Adjacency List Graph Implementation
Adjacency list of vertex 0
 head -> 19-> 20-> 10
Adjacency list of vertex 1
 head -> 14-> 5-> 13
Adjacency list of vertex 2
 head -> 15-> 0-> 20-> 4-> 7
Adjacency list of vertex 3
 head -> 7-> 12-> 20-> 3-> 5-> 9-> 11
Adjacency list of vertex 4
 head -> 1-> 17-> 3-> 6-> 13-> 8
Adjacency list of vertex 5
 head -> 11-> 6
Adjacency list of vertex 6
 head -> 2-> 13-> 7-> 3
Adjacency list of vertex 7
 head -> 3-> 18-> 10-> 11
Adjacency list of vertex 8
 head -> 12-> 1
Adjacency list of vertex 9
 head -> 6-> 20-> 10-> 1
Adjacency list of vertex 10
 head -> 9-> 4-> 7-> 20-> 10-> 12-> 15
Adjacency list of vertex 11
 head -> 16-> 10-> 17
Adjacency list of vertex 12
 head -> 1-> 14-> 17-> 12-> 9-> 11
Adjacency list of vertex 13
```

```
 head -> 9-> 2-> 10
Adjacency list of vertex 14
 head -> 3-> 19-> 12-> 1-> 13-> 0
Adjacency list of vertex 15
 head -> 4-> 19-> 5-> 3-> 2-> 18
Adjacency list of vertex 16
 head -> 1-> 10-> 17-> 7
Adjacency list of vertex 17
 head -> 18-> 1-> 8-> 5-> 15
Adjacency list of vertex 18
 head -> 1-> 0-> 9-> 6-> 10-> 3-> 18
Adjacency list of vertex 19
 head -> 7-> 13
Adjacency list of vertex 20
 head -> 14-> 13-> 1-> 20-> 9-> 10
Breadth-First Algorithm Time: 3.117e-06
Depth-First Algorithm Time: 5.781e-06
```

3.2 Commenti

L' efficienza degli algoritmi sembra esser influenzata molto da quella che è la grandezza del grafo, cioè il numero di nodi, e relativamente per quelli che sono i collegamenti tra di essi.
Per configurazioni piccole il Depth-First supera addirittura l'ordine di grandezza dei microsecondi, raggiungendo i 10^-7 secondi, mentre il Breadth-First soffre decisamente di più, specie nel caso della matrice di adiacenza, dove sembra essere decisivo l'utilizzo della coda.
Aumentando il numero di nodi si nota che i tempi tendono a livellarsi e dalle prove svolte, una volta raggiunti i 50 nodi si raggiunge un punto di equilibrio tra i tempi di esecuzione di tutti gli algoritmi di esplorazione in cui nessuno riesce a toccare l'ordine di grandezza dei microsecondi per tempo effettivo di esecuzione.
La prova con il maggior numero di nodi, 103, ha confermato questo equilibrio, mostrando un'aumentare proporzionale di tutti i tempi, tranne per quello del Breadth-First della lista di adiacenza, che a differenza degli altri algoritmi che esibiscono tempi dell'ordine di grandezza di 10^-4, mantiene un 10^-5, di poco superiore rispetto alle prove fatte con la metà dei nodi.
Queste ultime prove mettono in luce la migliore efficienza del Breadth-First rispetto al Depth-First, descritta nella letteratura, quando ci si trova ad affrontare strutture dati di una certa portanza.
Realisticamente il Depth-First che offre tempi migliori con piccole strutture non è poi confrontabile con il Breadth-First in quanto difficilmente algoritmi di ricerca saranno applicati su grafi della grandezza di 10 nodi.
Il passo successivo è l'implementazione di algoritmi di ricerca come l'$A*$, per avere un confronto reale su quella che è la complessità computazionale dei due algoritmi e i loro tempi effettivi di esecuzione in un problema utilizzato in diversi ambiti quali database implementati con grafi, che rappresentano l'essenza di questo problema.